Testtool User's Manual

STTST

PRELIMINARY DATA

1 Change History

Date	Version	Reason and Change
March 1999	1.0	Initial release
Jan 2000	1.1	New commands : pollkey, getkey, getstring, search_next
Mar 2000	1.2	Improvment : display, notation
Mar 2000	1.3	Add info about starting and stopping
Nov 2000	1.4	Add info about symbols ; correction of little mistakes
05 Oct 2001	1.5	Add 'ELIF' control statement.
Feb 2006	1.6	Add description for new commands: speek, spoke and unsource
Sept 2006	1.7	Add description for new commands about Profiling.
Nov 2006	1.7	Add 'EVAL' testtool command.

2 Summary

The Testtool software utility has been written to allow users to test drivers running on ST components. Testtool is a command interpretor. The users can add his own commands, in order to test them easily, and can make command sequences for automatic tests.

The general features of the Testtool environnement are as follows:

- Interactive command execution with flexible parameter defaulting action
- Explicit friendly syntax error reporting
- Consistent help capability for commands
- Support of symbolic constant, variable and expression evaluation for integers, floting point numbers and strings
- Command macro definition capability with flow control and looping constructs
- File I/O for command logging, command file execution and saving state

The capabilities of the inbuilt Testtool commands are described more fully in the following sections.

3 Commands & Functions

3.1 Starting and stopping

When Testtool is launched, the user must at first type on any key (*). Then a prompt is displayed. At the prompt, the user can enter any command according to this manual.

To quit Testtool, enter END or EXIT at the keyboard. The user is again required for a key (*). Press on Backspace key to exit, or press another key to go again in Testtool.

(*) This mechanism was implemented in an old release to allow the user to choose between DCU and UART interface ; this choice is unavailable since STTBX driver is used by Testtool.

3.2 Language Basics

The Testtool implements an interpreted language environnement - that is the lines of the language are analysed and executed individually as entered. This is some'what slower than the used of a compiled language at execution time but offers many advantages for an interactive environnement. The general characteristics of the language, and therefore the interactive "feel" of the tool, are defined by the interpreter engine that accepts and tokenises the lines of input. Some of these characteristics are important in understanding the synthax that the language features possess and the errors that may be encountered when using the tool. These general features are described in the sections below.

3.2.1 Statements

Executable language consists of a series of commands, assignments or control flow statements. Commands can be built in - i.e. the code that determines their action has been compiled and linked with the interpreter - or they can be macros, which are simply parameterised collections of other language statements. No terminator is required for a language statement - they are all assumed to be contained on one line and entry of a carriage-return character terminates the statement and begins the interpretation and execution.

Comments can be included in streams of commands by using the ":" character to indicate the end of a legal statement and the beginning of a comment. All tokens in the statement must be clearly distinguishable from the comment character, usually by employing white space. A special case o this, of course, is a comment only line where "," is the first character.

3.2.2 Delimiters

White space on a line (spaces and CR/LF) is significant to the Testtool and generally used to delimit components of the command statements. The number of spaces between components is not significant. In particular, any number of spaces at the beginning of the language statement will not affect its meaning or interpretation. Nor will the number of spaces between parameters within the statement. Other command or parameter delimiters that can be used interchangeably are commas (,) and backslash (\) characters. The number of these used within a statement is significant and can be used, for instance, to introduce place-holders into command statements. For instance, the statements:

```
> SOURCE "DEFAULT.COM" TRUE
> SOURCE, TRUE
```

are equivalent since the default filename for a set of commands to be "sourced" is "default.com". To indicate that this default value should be used, a comma is used to delimit a null token for this first positional parameter whilst explicitly giving the non-default value (echo of input lines) for the second parameter.

3.2.3 Defaulting

To ease the use of the language in an interactive mode, a general policy of defaulting has been adopted. Typical command statements will have a set of position sensitive parameters with the most frequently used parameters appearing first. The user can then specify only these parameter values which are explicity required and omit others where upon they will default to "sensible" values defined by the creator of the command. In particular, well written command statements will often default to the most general mode of usage or accept a default value that has been explicitly set by a previous statement.

3.3 Constants, Variables and Expressions

The implementation of full programming language features has been side stepped in the Testtool command language, but there still exist a powerfull underlying mechanism for mathematical manipulation and expression handling.

3.3.1 Numbers

The Testtool supports the use and manipulation of integer and floating point numbers. The number range of integers is limited by the supporting machine's natural 32 bit representation (in addition the value of the maximum negative integer in this representation is used internally to signifiy an error). For more precision in calculation, floating point types can be used. Constants and variables of both types can be declared used freely. Integrated usage is by far the most common in practice, and integers be used at any point where a value is required in the language.

Integer constants can be expressed in hexadecimal, decimal, octal or binary notation. At any given time, a default base for conversion of integrer numbers is in force. The default base conversion can be overriden by explicitly prefixing the number with a symbol that indicates the intended base of the constant value. A floating point value is signified by including a decimal point in the notation and can be given with no decimal places, or any number of digits after the decimal point. There are no base conversion issues with floating point numbers.

```
40 - INTEGER
h4F - HEX INTEGER
o67 - OCTAL INTEGER
b010 - BINARY INTEGER
+2 - DECIMAL INTEGER
-435 - DECIMAL INTEGER
4.0 - FLOATING POINT
0.2 - FLOATING POINT
3.2E5 - FLOATING POINT
```

To keep compatibility with previous releases, the following conventions remains available :

```
#4F - HEX INTEGER
$10 - BINARY INTEGER
```

The expression `b10+o10+10+h10` is equivalent to `2+8+10+16`.

3.3.2 Numeric Expressions

In place of any explicit constant value, a mathematical expression of arbitrary complexity can be used. The expression must not contain embedded spaces, since these might be interpreted as value terminators, but may include brackets to make precedence clear. Precedence will be exercised from right to left in normal evaluation. The available mathematical operators are:

- + integer and floating point addition
- - integer and floating point subtraction

- * integer and floating point multiplication
- / integer and floating point division
- % integer remainder
- & bitwise AND
- | bitwise OR
- ^ bitwise XOR
- ~ bitwise NOT (this is a unary operator)
- ! logical NOT (this is a unary operator)

Examples of the use of these operators in the context of expressions are given below:

```
23+44.0
1002*3
24/(34+2)
(23-22)*(400.0+0.1)
```

Where a floating point value is included inan expression, the whole resultant value becomes a floating point value, otherwise integer evaluation is assumed.

Note that there is no facility to include function calls inside an expression - this is precluded by both syntax and convertion since the language is purely procedural.

3.3.3 Numeric Symbols

In place of any constant number or expression a symbol can be used. Symbols can contain constant or variable numeric values and can be used freely within the language. Constant symbols are established by the programmer and predeclared within the interpreter - this is used for command constants such as TRUE, FALSE and PI. Symbol variables can be predeclared or freely created and deleted by the interactive user. A symbol is declared using the assignment statement as shown here:

```
APPLES = 23
ORANGES = 0.1
```

The variable symbols are declared and are set to the initial values given using this C style syntax. Thereafter the symbol can be used by referencing either its full or abbreviated name.

```
APPLES/34.2
ORANGES+ APPLES-1
(OR*APP)-22
```

Symbols can be used eiher as variables are used in programming languages or simply as a meaningful name for a commonly used numeric parameter value. In this way context specific constants, for instance, can be abbreviated or made symbolic (e.g. the value for a particular bit patter in a register). The rules for choosing symbol names are typical of many programming languages - no leading numeric characters are allowed, only alphanumeric digits and underscore characters can be used within names and only 32 characters in any name are significant.

The current know symbols and their values can be displayed using the utility "show" command.

Caution : if symbols B, O, and H are created by the user and set to a positive value, the expression B+O+H is interpreted as b0+o0+h0 = 0+0+0 = 0 !

3.3.4 Assignments

A variable assignment can be used to hold the results of an expression evaluation. The syntax of the assignement is that of C, and the target of the assignment may or may not already have been declared. Since some symbols may have been created as constants, not all assignments can succeed even in the case where the syntax is correct.

```
APPLES-ORANGES+(4/1-2)
BORED-TRUE
TRUE = 3 (WILL FAIL WITH AN ERROR MESSAGE)
```

Commands and command macros can and do return simple values of various sorts and thus can be used within an assignment statement. Due to limitations of the syntax, it is not possible to mix commands and generalised expressions so the format is limited to statements of the type:

```
TEMP=PRINT "A" 1 2 3
A123
SHOW TEMP
TEMP: "A123" - STRING VARIABLE
```

The type and value of any returned data from commands should be documented with the command.

3.3.5 String Symbols

Variables can also be created to hold string values. The strings can be used by commands as parameters where appropriate. Identical rules apply about declarations and usage, but the contents of a string are case preserved and can contain any characters. All string constants are represented by enclosing text within double quotes.

```
CARD="THE STRING IS CARD"
PAPER="THINNER THAN CARD"
```

The double quotes are necessary to identify a string constant, since all other features of the commands syntax are the same.

The length of a string is limited to 80 characters (quote + 77 characters + quote + null).

3.3.6 String Assignment

Once a string has been declared it may be reassigned freely and concatenated with other string using the "+" operator.

```
TEMP="HELLO"
TEMP=TEMP+"WORLD"
```

The quote character may be included within a string by proceeding it with the escape "\" character.

3.3.7 Predefined values

The following constants and variables are predefined :

```
HEXADECIMAL: 16 - integer constant
DECIMAL: 10 - integer constant
OCTAL: 8 - integer constant
BINARY: 2 - integer constant
TRUE: 1 - integer constant
FALSE: 0 - integer constant
PI: [fp] - floating point constant
COMMAND_FILE: "default.com" - string variable
LOG_FILE: "default.log" - string variable
```

```
ZERO: 0 - integer constant
```

3.4 Abbreviations

A name abbreviation and matching sheme is used which applies to commands, control flow statements, symbols and macro names, as follows:

Case is not significant in matching any name.

Any number of charachers (greater than two) that uniquely matches the symbols name can be used to reference the symbol.

In the case of ambiguity, the symbol name which most closely matches the given set of characters is used - i.e. the matched symbol is the one with the shortest of the matching declared names.

Where multiple variable types can be accepted, the possible variable names are searched in the order integer, floating point, string. Thus abbreviations which could match variables of more than one type will match the first appropriate definition found.

The first and most common use of such abbreviations is in the quick execution of commands themselves. Given a command with a defined full name of "show" the following abbreviations are equivalent and will invoke the command:

```
> SHOW
> SH
> SHO
> SHOW
```

Similar rules apply to parameters of commands, in particular symbols. For example, given the synbols "fred" and "fredsMum" have been created, the following abbreviated references are valid:

```
NAME TYPED  SYMBOL MATCHED
FR          FRED
FRED        FRED
FREDS       FREDSMUM
FREDSMUM    FREDSMUM
```

This abbreviation strategy has been designed to give flexibility, especially in interactive use, to reference fully descriptive names by some sub-string. Almost all situations of abbreviated reference wil be resolved un-ambiguously. This includes usage of symbol names declared as substrings of other symbol names such as above. However, if a symbol "fredsDad" is created, then the abbreviation "freds" will match either "fredsDad" or "fredsMum" depending on the order of declaration, since they are both the same length, in general, when coing macros and command scripts where the context of their use is not well defined, specify the full name of a symbol to be sure of matching the correct value.

3.4.1 Error Reporting

Syntax errors from all command are reported as they are interpreted and, on finding an error, the interpretation and execution will be aborted. Errors are reported as meaning fully as possible with an explicit identification of the position of the error on the supplied line. The positional information is given by feeding back to the user the state of the parsing process at the time the error was found. This is normally extremely accurate and visually unambiguous. All macro and control flow execution will be terminated by any such error, even at a nested level. For example:

```
> HELP GG
HELP GG
.....^.^
```

```
UNRECOGNISED COMMAND OR MACRO
> GG = 45*FG
GG = 45*FG
....^.....^
UNRECOGNISED ASSIGNMENT STATEMENT
```

All commands that find errors in their parameter values should reprot the problem in the same consistent manner.

4 Intrinsic Commands

The basic operation of the Testtool interpreter is supplemented by the inclusion of some basic command facilities that perform utility functions. The following descriptions give some details of these commands and their use. These intrinsic commands are, for the most part, idealised examples of the type of behaviour that should be found in all commands.

4.1 Close

```
CLOSE
```

The command CLOSE is used when commands are logged : it stops the logging, and closes the log file previously opened with the LOG command. It has no effect if no log file is currently opened.

```
> CLO
LOGGING STOPPED AND LOG FILE CLOSED
```

4.2 Delete

```
DELETE <SYMBOLNAME>
```

DELETE is a command used for removing symbols and macro definitions from the current environment. All symbols declared in the scope of a macro execution are deleted when it exits so this command will normally only be used in an interactive mode. Use the SHOW command to establish the current symbol table contents and the DELETE command to remove one or more selected variable symbols. Constants and command cannot be deleted. For example:

```
> SHOW AREA
1: AREA (RADIUS) - COMMAND MACRO
>> END RADIUS*RADIUS*PI
>> DEL AREA
> SH BASE
1: BASE: -214748 - INTEGER VARIABLE
> DEL BASE TRUE
..................^...^
CANNOT DELETE FIXED SYMBOL OR COMMAND
```

The command DEL X* can be used to delete all symbols starting by the letter X.

4.3 Eval

```
EVAL <COMMANDSTRING>
```

EVAL is a command used for executing a string content, as if it was entered from the testtool command prompt. This is usefull when the command to execute is hold by a variable :

```
> TRACECOMMAND=”PRINT ”
> MSG=”HELLO”
> EVAL TRACECOMMAND+MSG
HELLO
```

4.4 Exit or End

```
EXIT
END <VALUE>
```

EXIT, as a simple command, is used to terminate the operation of the Testtool when working interactively. EXIT, as a command statement is used to terminate the entry of command macro bodies and to specify any return value that might be required. EXIT and END are equivalent and can be used interchangeably. When a value is given in an EXIT or END statement in a macro definition, this value can be assigned to a variable by the caller of the macro. Note that this value can also be in the form of an expression. More details are given in the description of macros.

4.5 Getkey

```
GETKEY
```

The GETKEY command is used to get keyboard input : the command interpretor waits for a key hit. Then the prompt is displayed again. The key value is also returned as an integer value, and can be obtained like described in the example. This is useful for making interactive macros.

> KEY=GETKEY

Then if the user press on A key and want to know the key value :

```
> SHOW KEY
1: KEY: 97 (h61) - INTEGER VARIABLE
```

4.6 Getstring

```
GETSTRING
```

The GETSTRING command is used to obtained keyboard input. When launched, the user must press one or many keys. The input must be ended by the Enter (or Return) key. Then the prompt is displayed again. The input is also returned as a string, like described in the example.

> MSG=GETSTRING

Then if the user enter HELLO, press Return and want to know the key value :

```
> SHOW MSG
1: MSG: "HELLO" - STRING VARIABLE
```

4.7 Help

```
HELP <COMMANDNAME>
```

The HELP command provides an on-line and rapid aid-memoir facility for the language environment. If a particular command name is given as a parameter (in full or abbreviated form) then a (usually) one line summary of that commands function and parameters are given. This is not intended as a substitue for full documentation of the command function but a quick and easy way to remember parameter order and type. For example:

```
> HELP HELP
HELP - <COMMANDANT> DISPLAYS HELP STRING FOR NAMED
COMMANDS AND MACROS
> HE SHOW
SHOW - <SYMBOL NAME> DISPLAYS SYMBOL VALUES AND MACRO CONTENTS
```

Where no command name is given as a parameter, the summary of all command help lines is displayed, along with the definition lines of any macros that have been created.

```
> HELP S*
SHOW          - <symbolname> Displays symbol values and macro contents
SOURCE        - <filestring><echoflag> Executes commands from named file
SAVE          - <filestring><constflag> Saves macros & variables/constants
to named file
SPEEK         - <address> Extracts 16 bit memory value
SPOKE         - <address><value> Sets 16 bit memory value
SEARCH        - <data><address><range><max> Searchs data in local memory
    default: data=DATAVALUE addr=BASEADDRESS range=RANGE max_occurences=1
             the 1st match address is returned (0 if not found)
SEARCH_NEXT   - Continues the previous search in local memory
```

The command is not case dependant : HELP s* and HELP S* provide the sames results.

4.8 History

HISTORY <NUMBER>

The command HISTORY displays the list of the last commands which have been launched. For each command displayed, a number is given. When this number is given as a parameter, the corresponding command is launched.

```
> HISTORY
0   --
1   -- PRINT "HELLO !"
2   -- HELP SH*
> HI 1
PRINT "HELLO !"
HELLO !
>
```

4.9 Log

```
LOG <FILESTRING> <OPTIONS>
```

This command allows the recording of streams of interactive command statements, and optionally their resultant output, to ASCII files. The filename is a string parameter and therefore must be given surrounded in double quotes. If no name is given a default string variable LOG-FILE is used as the source for the filename. Any other string variable or constant can of course be used on the command line to specify the file.

The options is a string parameter, surrounded in double quotes, and should contains 1 or 2 characters like W, A, I, O, R. Using W, a new log file will be created (if any, the old one is deleted). Using A, the old log file is kept, and the data will be appended at the end of it. Using I, only the input commands will be logged. Using O, the input commands and the output (print command) will be logged. The specific option R is used to log only the output result (print commands containing the keyword “result”). If incompatible criterias are choosen at the same time,

like “WA” or “IO”, only the second one will be taken into account. If more than 2 options are required, an error message will occur, no log will be done . Unknown criterias are ignored. The default options are “WI”. If the command is runned twice, the second will close the first file before opening the second one. Examples:

```
> SHO LOG_
0: LOG_FILE: "DEFAULT.LOG" - STRING VARIABLE
> LOG
LOGGING INPUT TO FILE "DEFAULT.LOG" (MODE=W)
> LOG "RECORD.FILE" TRUE
CLOSE LOG FILE
LOGGING INPUT AND OUTPUT TO FILE "RECORD.FILE" (MODE=W)
> LOG LOG_FILE "AR"
CLOSE LOG FILE
LOGGING OUTPUT RESULTS TO FILE "DEFAULT.LOG" (MODE=A)
```

For compatibiltiy with old release, the options can replaced by an ouputflag :

```
LOG <FILESTRING> <OUTPUTFLAG>
```

If the outputflag is FALSE (the default value), commands will be logged in a form that could be used for subsequent execution via a SOURCE command and none of the resulting output will be recorded. If the output flag is specified as TRUE, all command statements in the logfile will be prefaced by the prompt and all command output will also be recorded. In this case the logfile acts as a faithful representation of the Testtool session as a whole.

4.10 Pollkey

```
POLLKEY
```

The POLLKEY command is used to test, without waiting, a hit on the keyboard. The key value is also returned as an integer value (0 if not hit detected), and can be obtained like described in the example. This is useful for making interactive macros.

```
> KEY=POLLKEY
> SHOW KEY
1: KEY: 0 (h0) - INTEGER VARIABLE
```

If the command is used in a macro and if a key is pressed, the macro will stop immediatly.

```
> DEFINE MYMACRO
FOR I 1 1000
POLLKEY
END
PRINT "DONE"
END
>
```

Launch MYMACRO, then press a key. The message “done” will not appear.

```
> MYMACRO
>
```

4.11 Print

```
PRINT <VALUES>
```

The PRINT command allows formatted output to be constructed from within the Testtool environment. The output of the PRINT command is a string containing formatted values in the order presented in the command statement. No intervening spaces are inserted. All types of values, symbols and expressions are permitted and any number can be put into a single statement. The output format of integer values will be affected by the default I/O radix value. For example:

```
> PR "VALUE " DECIMAL
VALUE 10
```

The value returned by the command in an assignment statement is the string that is printed, which allows multi level formatting to be performed if desired.

```
STRING = PRINT "DEFAULT LOG FILE IS "LOG_FILE
DEFAULT LOG FILE IS DEFAULT.LOG
> PR STRING "AND" COMMAND_FILE
DEFAULT LOG FILE IS DEFAULT.LOG AND DEFAULT.COM
>PRINT (8+2)*h10
160
>PRINT "Total=" 13*10 " Dollars"
Total=130 Dollars
```

4.12 Profiler_Init

```
PROFILER_INIT
```

The PROFILER_INIT command is used to initialize a Profiler. A Profiler can be used to analyze the performance characteristics of a target system.

Profiling is the term used to describe the gathering and subsequent analysis of a system's performance data. The profiler gathers information about a system by sampling the program counter (PC) at regular intervals. The acquisition starts when the command PROFILER_START is called. The results are stored in a log file when the command PROFILER_STOP is called. This report identifies where the processor spent its time during the period being profiled.

NB: For the moment only OS21 Profiler is implemented.

4.13 Profiler_Start

```
PROFILER_START
```

The PROFILER_START command is used to start the sampling of the program counter. When this command is called, the Profiler starts gathering information.

4.14 Profiler_Stop

```
PROFILER_STOP
```

The PROFILER_STOP command is used to end the gathering of system information. The Profiler stops sampling the program counter and copies all the samples collected to a file called "profiler..log" This file is available in the directory containing the Executable.

The log file produced by OS21 profiler can not be read directly. It should be interpreted by a Perl Script called "***os21prof.pl***" (you should have a Perl interpreter installed on your machine). This Script is available in the bin directory of the toolchain: \STM\ST40RX.X.X\bin\os21prof.pl

It is invoked as follows:

perl -w os21prof.pl <executable_file> <profile_file>

The following example displays a profile report for the application 'test.out', from

the profile data held in 'profile.log'.

perl -w os21prof.pl test.out profiler.log

For an easier use, it it recommended to use a Batch file. Here is an exemple of Batch for the video application called "vid_test.exe":

```
@echo off
REM
REM Purpose: "Profile.bat" parses the content of a OS21 Profiler Log File.
REM
REM Usage  : Drag and Drop a Profiler log file.
REM
REM NB     : "vid_test.exe" should be in the same directory
REM     than the log file.
REM

if not exist "%1" goto ShowUsage
if not exist "%~dp1vid_test.exe" goto ShowUsage
perl -w %ST40ROOT%\bin\os21prof.pl "%~dp1vid_test.exe" %1
goto Exit

:ShowUsage
echo *************************
echo * ERROR!
echo * Drag and Drop a Profiler log file (ex: "profile.log").
echo *
echo * The file "vid_test.exe" should be present in the same
echo * directory than the log file.
echo *************************
goto Exit

:Exit
@pause
```

To use this Batch file, simply Drag and Drop the log file (under windows explorer) and you will get the Profiling results displayed.

Example of result output:

```
OS21 profile analysis for T:\dvdgr-prj-stvid\tests\src\objs\ST40\vid_test.exe
(2 bytes per bucket, sampled at 1003 Hz, system wide profile)

  12.201 seconds (100.00%) :  Total duration

Task breakdown:
  11.154 seconds ( 91.41%) :  Idle Task
   0.314 seconds (  2.57%) :  STVID[0].Displa
   0.239 seconds (  1.96%) :  STVID[0].Decode
```

```
   0.219 seconds (  1.80%) :  VideoInject
   0.065 seconds (  0.53%) :  trace_buffer
   0.061 seconds (  0.50%) :  STVID_Injecter
   0.006 seconds (  0.05%) :  Root Task
   0.003 seconds (  0.02%) :  STVID[?].FmdTas
   0.003 seconds (  0.02%) :  STLAYER_Video
   0.001 seconds (  0.01%) :  STVID[0].TrickM

Interrupt level breakdown:
   0.105 seconds (  0.86%) :  Interrupt level 7
   0.007 seconds (  0.06%) :  Interrupt level 12
   0.025 seconds (  0.20%) :  Interrupt level 13

Symbolic breakdown:
   6.217 seconds ( 50.96%) :  _md_kernel_sleep
   4.936 seconds ( 40.46%) :  _scheduler_idle
   0.172 seconds (  1.41%) :  _md_kernel_intr_restore
   0.142 seconds (  1.16%) :  _md_kernel_intr_mask
   0.110 seconds (  0.90%) :  memcpy
   0.053 seconds (  0.43%) :  UART_InterruptHandler
   0.046 seconds (  0.38%) :  DecodeTaskFunc
   0.042 seconds (  0.34%) :  ComputeAndApplyThePolyphaseFilters
   ...
```

4.15 Save

```
SAVE <FILESTRING> <CONSTANTFLAG>
```

This command allows the saving of the state of the Testtool, in terms of symbols and macro definitions, into ASCII files. These files are identical to those read by the SOURCE command and therefore can be used to incrementally develop an enhanced Testtool environment tailored to the preferences of an individual user. The constantflag parameter allows optional saving of the constant values defined by the implementation of the Testtool. This is not normally useful because the resultant file, if used within a subsequent SOURCE command, will cause errors due to the redefinition of constant values.

It is important to realize that the filename is a string parameter and therefore must be given surrounded in double quotes. If no name is given a default string variable COMMAND_FILE is used as the source for the filename. Any other string variable or constant can of course be used on the command line to specify the file. For example:

```
> SHOW COMM
0: COMMAND_FILE: "DEFAULT.COM" - STRING VARIABLE
> SAVE
SAVING TO FILE "DEFAULT.COM"
> SAVE "MACRO.FILE" TRUE
SAVING TO FILE "CONSTANTS.FILE"
```

4.16 Show

```
SHOW <SYMBOL NAME>
```

This utility command allows the display of the contents of the symbol table in a controllable format and can be used for interactive use of for debugging of macros or command scripts. If a valid symbol name is given as a parameter, the type and value of the symbol, if recognised. If the symbol is a macro definition, than its contents are shown. For example:

```
SHOW PI
0: PI: 3.14156 - FLOATING POINT CONSTANT
```

```
> SHOW TRUE
0: TRUE: 1 (h1) - INTEGRER CONSTANT
> SHOW AREA
1: AREA (RADIUS) - COMMAND MACRO
> RESULT = RADIUS*RADIUS*PI
> END RESULT
```

In this display, the value to the left of the symbol name is the nesting depth at which the variable was declared. This is useful to identify situations where scope rules have caused one symbol to be obscured by another more locally declared. Where no symbol name parameter is given to the SHOW command, the full contents of the symbol table are displayed in order or declaration.

4.17 Source

```
SOURCE <FILESTRING> <ECHOFLAG>
```

This command allows the execution of streams of commands that are held in ASCII files. The streams of commands can be echoed as they are input and executed using the optional flag which defaults to FALSE. There is no restriction on the commands that can be contained in the files, but they are often used to define macro commands which can later be executed in interactive mode.

It is important to realize that the filename is a string parameter and therefore must be given surrounded in double quotes. If no name is given a default sting variable COMMAND_FILE is used as the source for the filename. Any other string variable or constant can of course be used on the command line to specify the file. For example:

```
> SHOW COMM
0: COMMAND_FILE: "DEFAULT.COM" - STRING VARIABLE
> SOURCE
INPUT FROM FILE "DEFAULT.COM"
> SOURCE "MACRO.FILE" TRUE
-> DEFINE AREA RADIUS
-> END RADIUS*RADIUS*PI
```

4.18 Stat

```
STAT
```

The command STAT provides some statisticals about memory usage : the number of existing symbols, and the free local memory size.

```
> STAT
declared symbols  = 204 (162 commands + 0 macros + 2 strings + 40 numbers)
allocated symbols = 204 / 1100
free memory size  = 347500 bytes
```

4.19 Unsource

```
UNSOURCE <FILESTRING> <ECHOFLAG>
```

UNSOURCE is a command used to remove commands that are held in sourced files. All symbols that are already defined in the given file are deleted. An error message is returned if symbols to delete are not defined. The streams of commands to be deleted can be echoed as they are input using the optional flag which defaults to FALSE.

The filename <FILESTRING> is a string parameter and therefore must be given surrounded in double quotes. If no name is given a default sting variable COMMAND_FILE is used as the source for the filename. Any other string variable or constant can of course be used on the command line to specify the file. For example:

```
> SHOW COMM
0: COMMAND_FILE: "DEFAULT.COM" - STRING VARIABLE
> UNSOURCE
Deleting Symblos from file "DEFAULT.COM"
> UNSOURCE "MACRO.FILE" TRUE
Deleting Symblos from file "MACRO.FILE"
-> DEFINE AREA RADIUS
-> END RADIUS*RADIUS*PI
```

4.20 Verify

```
VERIFY <TRUE | FALSE>
```

This command controls the automatic echoing of executed command statements to the screen. This can be useful when debugging macros as it allows you to see which lines are being executed, which line cause an error and so on. It can be turned on or off at any point. The command VERIFY without a parameter reports the current state of the echo flag. For example:

```
> VERIFY TRUE
COMMAND ECHO IS ENABLED
> D = AREA 5
-> END RADIUS*RADIUS*PI
> VER
COMMAND ECHO IS ENABLED
```

4.21 Wait

```
WAIT <MILLISECONDS>
```

The command WAIT is used to do a pause according to the given duration which must a number of milli-seconds. Notice that for a very short pause, the time needed by the command interpretation may be greater than the expected duration. The result depends also of the microprocessor speed: it will be right with 1 tick per 64 micro-seconds.

5 Memory commands

Some commands provides access to the contents of the local memory. They are described in this section. Some predefined variables are used by them :

```
BASEADDRESS: h80000000 - integer variable
RANGE: 256 - integer variable
DATAVALUE: 0 - integer variable
ADDRESSVALUE: h80000000 - integer variable
```

5.1 Bpeek

```
BPEEK <ADDRESS>
```

The command BPEEK is used to extract a memory value stored on 8 bits. If the address value is not given, the selected address will be the value of the variable BASEADDRESS.

5.2 Bpoke

```
BPOKE <ADDRESS> <VALUE>
```

The command BPOKE sets the memory with the given value at the specified address. It is a 8 bits value. If the parameters are not given, the selected data comes from the variables BASEADDRESS and DATAVALUE.

5.3 Copy

```
COPY <ADDRESS> <RANGE> <NEWADDRESS>
```

The command COPY is used to copy a range of memory data from the source address to the target address. If the parameters are not given, the selected data comes from the variables BASEADDRESS, RANGE and ADDRESSVALUE. Source and target addresses must not be equal,and the range must not be a negative value.

5.4 Display

```
DISPLAY <ADDRESS> <RANGE> <FILENAME> <OPTION>
```

The command DISPLAY gives a dump of the memory contents. If the parameters are not given, the selected values comes from the variables BASEADDRESS and RANGE. The data are displayed byte per byte, in hexadecimal mode; the translation into readable characters appears on the right; the address is displayed on the left.

```
> DISP
h80000000: 00 00 00 80  00 00 00 80  00 00 00 80  00 00 00 80  ................
h80000010: 00 00 00 80  00 00 00 80  00 00 00 80  00 00 00 80  ................
h80000020: 00 00 00 80  00 00 00 80  00 00 00 80  00 00 00 80  ................
h80000030: 00 00 00 80  00 00 00 80  00 00 00 80  00 00 00 80  ................
h80000040: 00 00 00 00  00 00 00 80  a4 fd 3f 40  00 fc 3f 40  ..........?@..?@
h80000050: 01 80 ff ff  01 00 00 80  44 23 38 40  38 92 1c 40  ........D#8@8..@
h80000060: 01 40 00 00  00 00 00 80  7c 13 2a 40  78 92 1c 40  .@......|.*@x..@
h80000070: f6 b6 8d f9  ee fa 99 d5  6a 5e d2 d5  e9 eb ee 1b  ........j^......
h80000080: ab d2 f5 46  57 d9 42 5f  ef 6f 69 35  9b 60 5d 66  ...FW.B_.oi5.`]f
h80000090: 7c c2 ce f2  5a 4f 5a 54  ad 33 6d 37  e7 3b da bf  |...ZOZT.3m7.;..
h800000a0: af cb 3e 2c  5f 65 fb d6  ef f5 9e 9f  c0 7e ad 4f  ..>,_e.......~.O
h800000b0: 53 f1 36 bf  db 80 79 a0  a4 fa 0d a7  82 7f 34 35  S.6...y.......45
h800000c0: 00 00 00 00  00 00 00 80  a4 fd 3f 40  00 fc 3f 40  ..........?@..?@
h800000d0: a6 ae 20 3e  a6 ce b2 d3  b6 e4 d4 f3  f6 59 dd 6e  .. >.........Y.n
h800000e0: 01 40 00 00  00 00 00 80  7c 13 2a 40  78 92 1c 40  .@......|.*@x..@
h800000f0: c7 ef ff 00  04 00 10 80  6c 1d 38 40  80 4b 1b 40  ........l.8@.K.@
h80000100: 88 e2 9f 3f  93 4e ed e1  a2 5e 9d a5  95 49 47 c8  ...?.N...^...IG.
```

If a file name is given, The dumped data can be stored in an ASCII file :

```
DISPLAY BASE 16 "DUMP.TXT" "W"
```

In order to append the data to an existing file,type :

```
DISPLAY BASE 16 "DUMP.TXT" "A"
```

The default option is “W”.

5.5 Fill

```
FILL <ADDRESS> <RANGE> <VALUE>
```

The command FILL is used to fill the memory with the specified value. All data starting from the given address are set to to this 32 bits value. If the parameters are not given, the selected data comes from the variables BASEADDRESS, RANGE and DATAVALUE. The range must be a positive value.

5.6 Peek

```
PEEK <ADDRESS>
```

The command PEEK is used to extract a memory value stored on 32 bits. If the address value is not given, the selected address will be the value of the variable BASEADDRESS.

5.7 Poke

```
POKE <ADDRESS> <VALUE>
```

The command POKE sets the memory with the given value at the specified address. It is a 32 bits value. If the parameters are not given, the selected values comes from the variables BASEADDRESS and DATAVALUE.

5.8 Search

```
SEARCH <DATA> <ADDRESS> <RANGE> <MAX>
```

The command SEARCH start a data research in memory. The data may be a string enclosed with double quotes, or an integer value. The data size is evaluated (it may 1, 2, 3 or more bytes), then the search is done according it, starting from the specified address. The "max" parameter is the maximum of expected occurences. The search is ended when the all the range has been parsed, or when enough occurences are found. If parameters are missing, the default data comes from the variables DATAVALUE,BASEADDRESS, RANGE and max. is set to 1.

```
> SEARCH
Search h00 on 1 bytes from h80000000 to h80000100 ...
Match found at h80000000
> SEA h0
Search h00 on 1 bytes from h80000000 to h80000100 ...
Match found at h80000000
> SEA h000080
Search h000080 on 3 bytes from h80000000 to h80000100 ...
Match found at h80000001
> SEA 128
Search h00000080 on 4 bytes from h80000000 to h80000100 ...
Match found at h80000000
> SEA "?@" BASEADDRESS RANGE 4
Search [?@] on 2 bytes from h80000000 to h80000100 ...
Match found at h8000004a
Match found at h8000004e
Match found at h800000ca
Match found at h800000ce
> SEA "D#8@8" BASEAD
Search [D#8@8] on 5 bytes from h80000000 to h80000100 ...
Match found at h80000058
```

The first match address is also returned (0 if not found). Example :

```
> X=SEARCH 128
Search h00000080 on 4 bytes from h80000000 to h80000100 ...
Match found at h80000000
> SHOW X
1: x: -2147483648 (h80000000) - integer variable
```

5.9 Search_next

```
SEARCH_NEXT
```

The command SEARCH_NEXT continues the previous search in memory. It starts immediatly after the location of the last occurence found. It stops when the the next occurence is found or when the range is reached. Nothing is done if no previous search was done.

```
> SEARCH h3f40
Search h3f40 on 2 bytes from h80000000 to h80000100 ...
Match found at h8000004a
> SEARCH_NEXT
Match found at h8000004e
> SEARCH_N
Match found at h800000ca
```

5.10 Speek

```
SPEEK <ADDRESS>
```

The command SPEEK is used to extract a memory value stored on 16bits. If the address value is not given, the selected address will be the value of the variable BASEADDRESS.

5.11 Spoke

```
SPOKE <ADDRESS> <VALUE>
```

The command SPOKE sets the memory with the given value at the specified address. It is a 16bits value. If the parameters are not given, the selected values comes from the variables BASEADDRESS and DATAVALUE.

6 Command Macros

The Testtool language is fully intended to be user extensible at both the programmer level and at the user level. At the programmer level this involves hard coded base routines which operate directly on programmatic structures. These base commands are set up and made available in a particular fashion according to the implementers idea of the context. These functions are usually created to be simple in form and atomic in operation. At the user level, when a more complex function is desired, or simply a base function with a different set of default parameters or options. It can be created by the user using macro definitions and thereafter used in an identical way to the original command. The feel of macro commands is very similar to that of the base commands even down to the defaulting and error reporting procedures. If this facility were simply an aliasing function it would be quite powerful, but limited by the demands of textual substitution. Instead it has been implemented as a full procedural language extension and offers a variety of control constructs to provide a usefully rich functional development environment.

6.1 Macro Definition and Invocation

A macro is normally defined by specifying, using the DEFINE command, a macro name and any number of format parameter names.

```
> DEFINE MULDIV A B C
```

The formal parameter names can be considered to be equivalent to symbol declarations which are in effect only during the execution of the macro body. The types of these formal parameters symbols are, however, undefined until the macro is called as explained below. Additional declarations of symbols can be made using assignment statement in the body of the macro as required. All of these symbols will be created on the fly as the macro is executed and will crease to exist when the macro exits. Statements within the macro will also be able to access global variables, that is variables declared in the enclosing scope of the macro whose names do not clash with the variables declared locally. The body of the macro can contain any command statements that are used elsewhere and the definition and execution of body statements is terminated using an EXIT or END statement. These two statements are exactly equivalent in function.

After entering the DEFINE statement, the Testtool expects the next sequence of line to be the body of the macro, up till and including a matching END or EXIT statement. The lines are not interpreted at this stage, simply stored along with the macro definition line. For example, the body of the macro defined above might be entered as:

```
>> RES =0
>> RES:=(A*B)/C
>> PRINT "THE RESULT IS" RES
>> EXIT
```

Executing the macro defined above is very simple:

```
> MULDIV 3 2 1
THE RESULT IS 6
> MU 5 4 2
THE RESULT IS 10
```

Note that the macro is invoked by simply giving its name, in full or abbreviated form, at the command line along with its parameters. Any value parameters given will be fully evaluated before being passed into the macro. If a string parameter is given to the macro above, a string symbol will be created under the name of the formal parameter and will probably result in an error when evaluating the internal expression. If a value is left unspecified on the invocation line, a numaric symbol is created for the formal parameter, with a value of zero.

6.1.1 Returning Results

Clearly the example given above has only a limited use and at some stage it will be necessary to return results from a macro execution. The could be achieved by storing the result in a global variable of course but this is inflexible and only allows operation of the macro in certain ways. The other alternative is to have the value calculated and returned into the calling scope by means of an assignment.

```
> DEFINE MULDIV A B C
>> NUM= (A*B)/C
>> EXIT NUM
```

At first sight this does not appear to achieve anything. However, when called, the target of the assignment in the macro invocation, the integrer variable “res” is given the value that results from the calculation within the macro and referenced in the END statement.

```
> RES= MULDIV 3 2 1
> SHO RES
6
```

Again, note that the use of macros is purely procedural in nature. It would be impossible to use a macro call in the context of an expression due to both the syntax involved (the use of spaces as parameter separators) and the fact that no function or command value concept is exhibited.

6.1.2 Control Flow

Macros are given real programming power with statements that control conditional execution. These are presented in a strictly structured fashion and, in fact, are implemented using a parameter free macro system themselves. The functions available are IF/ELSE, WHILE and FOR, and their formats are described on the following sections. The logical expression which controls the conditional statements can be formed as a comparison between two numeric values or a numeric expression which evaluates to a logical result. In this sense a value of zero equates to logical FALSE. The functions available for comparison are:

```
> greater than
< less than
== equal to
!= not equal to
>= greater or equal
<= less or equal
<> not equal to
```

The following illustrates the use of this comparison capability:

```
> IF A>=B
> IF A
> IF A<>B
> IF (A*2)/3>=1
```

Note that comparisons cannot be nested and that comparisons between floating point values will in fact be performed in the integer domain.

6.2 Intrinsic Macro Commands

The following sections describe the macro related command statements and flow control structures.

6.2.1 Define

```
DEFINE <MACRO> <PARAMETERS>
```

The DEFINE command allows the creation of command macro procedures. The macros possess a name and a number of formal parameters which are supplied on the definition line. The content of the macro is a sequence of command statements, declarations, assignments, etc.

There is no restriction on the contents of a macro, but the validity of the contents is only checked at execution time and any error will terminate execution. Macros can be defined recursively (be careful!) and can contain other macro definitions. Entry of the definition of a macro will terminate when the number of END (or EXIT) statements equals the number of DEFINE statements in the body (including the initial DEFINE statement).

Macro names are subject to the same restrictions as other symbols and additionally may not clash with existing command names or other macro definitions. If there is a macro defined in the current scope of the same name in an enclosing scope, the more global definition will be replaced by the new definition.

6.2.2 If and Else

```
IF <COMPARISON>
> STATEMENTS
```

```
> END
ELIF <COMPARISON>
> STATEMENTS
> END
ELIF <COMPARISON>
> STATEMENTS
> END
...
ELSE
> STATEMENTS
> END
```

The result of the logical expression is used to determine which of the parameter free macro bodies is executed. The body statements can be of any form, including other macro definitions and control flows, but must be terminated with an END or EXIT statement. There is no compulsion to match each IF / ELIF statement with a ELIF / ELSE clause, but clearly ELIF / ELSE cannot be used without a matching IF clause. Spaces are not available in a comparaison expression.

```
> IF A>=B
>> SHOW A
>> END
> ELSE
>> SHOW B
>> END

> IF (X|Y)&Z
>> PRINT "True"
>> END
```

6.2.3 While

```
WHILE <COMPARISON>
> STATEMENTS
> END
```

The execution of the macro body statements is controlled by the evaluation of the logical expression before each execution. The comparison must be formed from currently declared variables and expressions, at least one half of which is assumed to be changed in value by statements within the macro body. The macro body can contain other statements as desired including other control flows and macro definitions but must be terminated by an END or EXIT statement. For example:

```
> WHILE A<B
>> A:=A+1
>> END
```

6.2.4 For

```
FOR <CONTROL VARIABLE> <INITIAL> <FINAL> <STEP>
> STATEMENTS
> END
```

The execution of the body statements is repeated over the range of values given for the control variable. The step value defaults to unity (+1), but can be defined as being negative. The initial and final values default to unity as well (not especially useful). The control variable symbol name need not exist before the statement, but will exist

after the loop has terminated with an undefined value. The macro body can contain other statements as desired including other control flows and macro definitions but must be an END or EXIT statement.

```
> FOR I 1 10
>> A = A*A
>> END
```

6.2.5 Symbols in macro and statements

A new symbol can be created in a macro. After the execution of the macro, the symbol is deleted.

A symbol can also be created inside a WHILE ... END or IF ... END statement. After the END, the symbol is deleted.

More generally, the scope (or access) to a variable depends on the depth of a macro or statement. A symbol created at the prompt (high level), is always available. A symbol created in a low level can't be used at a high level.

The depth of a symbol can be known with the command SHOW.

6.3 Examples of Macro Usage

6.3.1 Debug Statements

In order to debug other macros, it is often useful to create small macros which give the values of critical variables during execution. This can be done using the PRINT statement, perhaps enabled by the value of some globally declared variable.

```
DEFINE DEBUG NAME VALUE
    IF DEBUGFLAG>1
        PRINT "VALUE OF " NAME " " VALUE
    END
END
```

It is also sometimes helpful to use the SHOW command to dump the values of all variables at critical points, especially in highly nested calling sequences.

6.3.2 Expressions in end Statement

A simple macro, used elsewhere in this text, illustrates the possible brevity in the creation of macros. To calculate the area of a circle given its radius, a first attempt might be:

```
DEFINE AREA RADIUS
RESULT = RADIUS*RADIUS*PI
END RESULT
```

But this can be easily and cleanly re-coded as:

```
DEFINE AREA RADIUS
END RADIUS*RADIUS*PI
```

Where all the calculation is performed as an expression in the END statement.

6.3.3 Factorials

As an example of the use of local variables and control flow, the following macro is useful. It illustrates the fact that formal parameters of the macro are always created as variables in the local scope of the macro execution (which is absolutely essentiel for the recursion) where as those variables declared by assignment will only be

created in the first scope encountered where they do not already exist. It also illustrates the use of nested macro decarations.

```
DEFINE FACTORIAL VALUE
RESULT = 0
DEFINE RECURSION NUM
IF NUM>1
    RESULT = RECURSION NUM-1
    RESULT = RESULT*NUM
    PR "NUM "NUM "RESULT " RESULT
END
ELSE
    RESULT = NUM
END
END RESULT
RESULT = RECURSION VALUE
END RESULT
```

Information furnished is believed to be accurate and reliable. However, STMicroelectronics assumes no responsibility for the consequences of use of such information nor for any infringement of patents or other rights of third parties which may result from its use. No licence is granted by implication or otherwise under any patent or patent rights of STMicroelectronics. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. STMicroelectronics products are not authorized for use as critical components in life support devices or systems without express written approval of STMicroelectronics.

The ST logo is a trademark of STMicroelectronics

2006 STMicroelectronics - All Rights Reserved

STMicroelectronics GROUP OF COMPANIES

Australia - Brazil - Canada - China - France - Germany - Italy - Japan - Korea - Malaysia - Malta - Mexico - Morocco
The Netherlands - Singapore - Spain - Sweden - Switzerland - Taiwan - Thailand - United Kingdom - U.S.A.